http://java.sun.com/developer/technicalArticles/glassfish/GFandMySQL_Part2.html Mar 07, 2009

Article

GlassFish and MySQL, Part 2: Building a CRUD Web Application With Data Persistence

By Ed Ort and Carol McDonald, November 2008

Article Index

This is the second article in a series of articles on GlassFish and MySQL. Part 1 of the series describes the advantages of using GlassFish with MySQL and illustrates why the combination is a perfect choice for developing and deploying web applications. In Part 2, you'll learn how to develop a create, read, update, delete (CRUD) web application that uses GlassFish and MySQL. The application uses the Java Persistence API implemented in GlassFish to manage data persistence.

An important characteristic of both GlassFish and MySQL is that they're easily integrated into popular development tools. For example, plug-ins are available for both GlassFish and MySQL to integrate them into the NetBeans IDE and the Eclipse IDE. In addition, NetBeans IDE 6.1 With GlassFish and MySQL Bundle Download is available that integrates GlassFish v2 Update Release 2 (UR2) and MySQL 5.0 Community Server into NetBeans IDE 6.1. You can also download GlassFish v2UR2 with either NetBeans IDE 6.1 or NetBeans IDE 6.5 in a single bundle. A precursor to the next version of GlassFish, called the GlassFish v3 Prelude, is also available with NetBeans IDE 6.5.

This article shows you how to use the NetBeans IDE with GlassFish and MySQL to create the CRUD application. Specifically, you'll take advantage of features in NetBeans IDE 6.5, GlassFish v2UR2, and MySQL 5.1 Community Server to build and deploy the application.

You can examine the completed CRUD application by downloading and expanding the petcatalog application package.

Contents

The Application

The application for this article allows users to search an online catalog of pets. For example, users can search for a specific type of pet, such as medium-sized dogs, and display information about the items of that type in the catalog. Figure 1 shows a page that the application displays with this type of information.

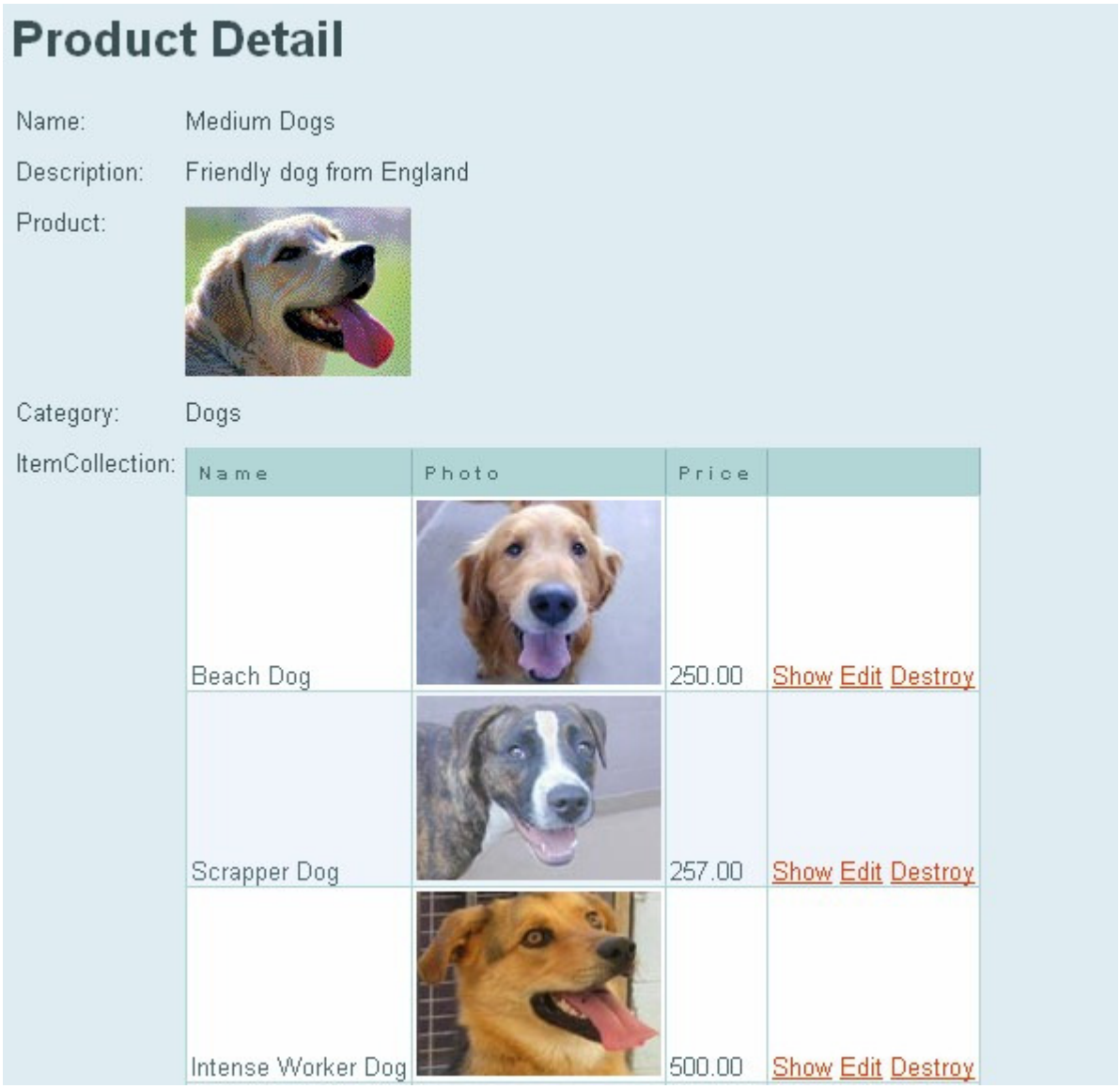

Figure 1. *Medium-Sized Dogs in the Catalog*

Retrieving information from the catalog represents the read or R capability of the CRUD application. As Figure 2 shows, a user can create a new item and add it to the catalog. This represents the application's create or C capability.

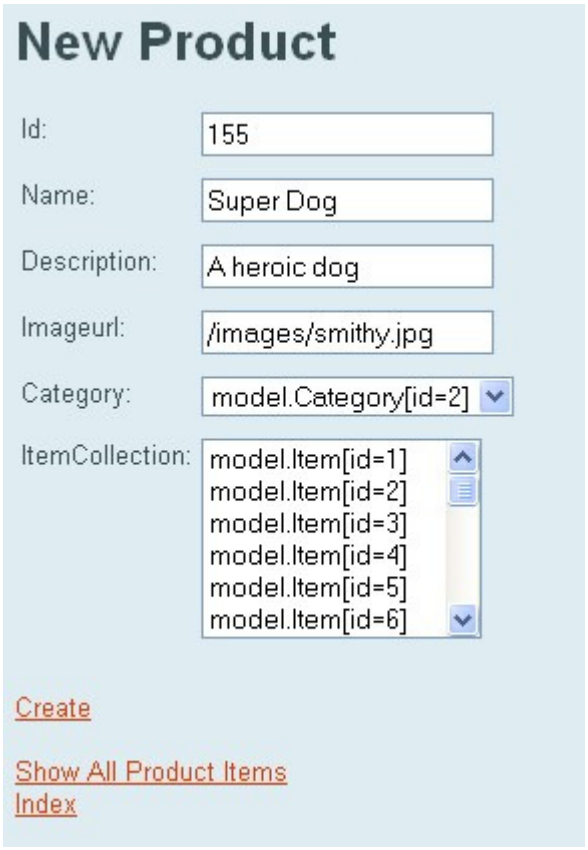

Figure 2. *Creating a New Item*

A user can also edit an entry in the catalog. This demonstrates the application's update or U capability. A user can also delete an entry. This demonstrates the application's delete or D capability.

Inside the Application

Before building the application, let's examine the components of the completed application. If you haven't already done so, download and expand the petcatalog application package.

The Catalog

Central to the application is the catalog, which is contained in a MySQL database named `petcatalog`. The `petcatalog` database has four tables:

- `category`. Contains information about each item category in the catalog, such as dogs. For each category, the table contains an identification (ID) number, name, description, and URL of a representative image.
- `product`. Contains information about each product type within a category, such as medium dogs or small dogs. For each product, the table contains an ID number, product ID number, name, description, and URL of a representative image.
- `item`. Contains information about each item in the catalog, such as beach dog. For each item, the table contains an ID number, product ID number, name, description, URL of the item's image, URL of the item's thumbnail image, price, and address ID.
- `address`. Contains information about the address of each item in the catalog. The address includes street, city, state, zip code, latitude, and longitude.

For example, here are the contents of the `category` table:

```
+----+----------+-----------------------------+---------------------------+
| id | name     | description                 | imageurl                  |
+----+----------+-----------------------------+---------------------------+
|  1 | Cats     | Loving and finicky friends  | /images/cats_icon.gif     |
|  2 | Dogs     | Loving and furry friends    | /images/dogs_icon.gif     |
|  3 | Birds    | Loving and feathery friends | /images/birds_icon.gif    |
|  4 | Reptiles | Loving and scaly friends    | /images/reptiles_icon.gif |
|  5 | Fish     | Loving aquatic friends      | /images/fish_icon.gif     |
+----+----------+-----------------------------+---------------------------+
```

You will find a file named `catalog.sql` in the `petcatalog` application . The file contains the SQL statements that create the tables and fill them with data. For example, here are the SQL statements that create the `category` table and fill it with data:

```
create table category(
    id BIGINT NOT NULL,
    name VARCHAR(25) NOT NULL,
    description VARCHAR(255) NOT NULL,
    imageurl VARCHAR(55),
    primary key (id)
);

INSERT INTO category VALUES(1, 'Cats', 'Loving and finicky friends', '/images/cats_icc
INSERT INTO category VALUES(2, 'Dogs', 'Loving and furry friends', '/images/dogs_icon.
INSERT INTO category VALUES(3, 'Birds', 'Loving and feathery friends', '/images/birds_
INSERT INTO category VALUES(4, 'Reptiles', 'Loving and scaly friends', '/images/reptil
INSERT INTO category VALUES(5, 'Fish', 'Loving aquatic friends', '/images/fish_icon.gi
```

This demonstrates how easy it is in MySQL to create database tables and fill them with content. You manage and query databases in MySQL using ANSI standard SQL, a language that many users find easy to use because of its relatively straightforward syntax.

It's easy to manage and query databases in MySQL using SQL.

Model-View-Controller

If you further examine the completed application, you'll notice that it follows the model-view-controller (MVC) design pattern. That is, the application isolates its data, known as the model, from the user interface or view, and from the code that manages the communication between the model and the view, known as the controller. You'll find the model and the controller in the `src/java` directory of the application. You'll find the view in the `web` directory.

Let's first look at the model for the application.

The Model

The model not only represents the data for the application, but it also represents persistent data, that is, data that persists beyond the life of the application. In other words, the model represents an application's persistent business domain objects. One of the advantages of using GlassFish as the application server for this application is that it includes a built-in persistence manager, the Java Persistence API based on Toplink Essentials. Taking advantage of this feature, the application uses the Java Persistence API to manage data persistence.

One of the advantages of using GlassFish as the application server for this application is that it includes a built-in persistence manager, the Java Persistence API.

If you open the `model` directory, you will see various Java classes. Four of these classes, `Address`, `Category`, `Item`, and `Product`, are entity classes -- typical Java Persistence API entity objects -- for the four tables in the `petcatalog` database. In the Java Persistence API, an entity instance, an instance of an entity object, represents a row of data in a database table. For example, `Item` is an entity class that maps to the `item` table in the `petcatalog` database. An instance of the `Item` entity represents a row of data in the `item` table. Here is part of the source code for the `Item` class:

```
package model;

import java.io.Serializable;
...

@Entity
@Table(name = "item")

public class Item implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "id")
    private Long id;
    @Basic(optional = false)
    @Column(name = "name")
    private String name;
    @Basic(optional = false)
    @Column(name = "description")
    private String description;
    @Column(name = "imageurl")
    private String imageurl;
    @Column(name = "imagethumburl")
    private String imagethumburl;
    @Basic(optional = false)
    @Column(name = "price")
    private BigDecimal price;

    @JoinColumn(name = "address_id", referencedColumnName = "id")
    @ManyToOne
    private Address address;

    @JoinColumn(name = "product_id", referencedColumnName = "id")
    @ManyToOne
    private Product product;

    //getters and setters
    ...
```

Notice the `@ManyToOne` annotations in the `Item` class. These specify that the `Item` class has a many-to-one relationship with the `Address` class and a many-to-one relationship with the `Product` class. This means that there can be multiple items in the catalog associated with the same address or with the same product type.

Conversely, the `Address` and `Product` classes have a one-to-many relationship with the `Address` class. This means that an item cannot be associated with multiple addresses or multiple product types. The one-to-many relationship is specified in `@OneToMany` annotations in the `Address` and `Product` entity classes. Here is the one-to-many annotation in the `Address` class:

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "address")
```

Here is the one-to-many annotation in the `Product` class:

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "product")
```

Annotations are an important Java EE 5 usability enhancement that is implemented in GlassFish. The Java Persistence API is only one of the technologies implemented in GlassFish that supports the use of annotations. In the CRUD application, annotations simplify the code for objects such as entity classes, enabling you to code these objects as Plain Old Java Objects (POJOs).

Annotations are an important Java EE 5 usability enhancement that is implemented in GlassFish. Using annotations in the Java Persistence API simplifies the code for entity classes, enabling you to code these objects as Plain Old Java Objects (POJOs).

The `model` directory also contains controller and converter classes. Each controller class invokes methods in Java Persistence API controller classes to perform operations for a corresponding entity class, such as creating and editing an instance of the entity class. Java Persistence API controller classes are in the controller layer of the application. For example, the following method in the `ItemController` class invokes the `create()`

method in the `ItemJpaController` class to create an instance of the `Item` class:

```
private ItemJpaController jpaController = null;

public String create() {
        try {
            jpaController.create(item);
            JsfUtil.addSuccessMessage("Item was successfully created.");
        } catch (Exception e) {
            JsfUtil.ensureAddErrorMessage(e, "A persistence error occurred.");
            return null;
        }
        return listSetup();
    }
```

The converter classes use JavaServer Faces technology (often referred to as JSF) to convert instances of the corresponding entity class to `String` objects, or the reverse -- `String` objects to entity classes. For example, the following method in the `ItemConverter` class converts an instance of the `Item` class to a `String` object:

```
public Object getAsObject(FacesContext facesContext, UIComponent component, String str
        if (string == null || string.length() == 0) {
            return null;
        }
        Long id = new Long(string);
        ItemJpaController controller = (ItemJpaController) facesContext.getApplicatior
        return controller.findItem(id);
    }
```

JavaServer Faces technology also plays an important role in the view layer of the application.

The Controller

The `controller` directory contains Java Persistence API controller classes. Each of these classes uses the Java Persistence API to handle operations for the corresponding entity class, such as creating and editing instances of the entity class. For example, here is part of the `create` method in the `ItemJpaController` that creates an instance of the `Item` entity class:

```
public void create(Item item) throws PreexistingEntityException, RollbackFailureExcept
        EntityManager em = null;
        try {
            utx.begin();
            em = getEntityManager();
            Address address = item.getAddress();
            if (address != null) {
                address = em.getReference(address.getClass(), address.getId());
                item.setAddress(address);
            }
            Product product = item.getProduct();
            if (product != null) {
                product = em.getReference(product.getClass(), product.getId());
                item.setProduct(product);
            }
            em.persist(item);
            if (address != null) {
                address.getItemCollection().add(item);
                address = em.merge(address);
            }
            if (product != null) {
                product.getItemCollection().add(item);
                product = em.merge(product);
            }
            utx.commit();
...
```

The View

The view layer uses data from domain objects provided by the controller to generate a web page. The view is rendered in the application using JavaServer Pages (JSP)

GlassFish implements Java EE

technology and JavaServer Faces technology. JSP enables the dynamic content required by the application, and JavaServer Faces technology provides UI components for the application. These are only two of a number of Java EE web technologies supported by GlassFish that provide simple, consistent mechanisms for extending web applications beyond static web pages.

web technologies such as JSP and JavaServer Faces technology that provide simple, consistent mechanisms for extending web applications beyond static web pages.

If you navigate to the `web/product` directory in the `petcatalog` application, you'll see a file named `List.jsp`. This file provides one example of how JSP and JavaServer Faces technology are used in the application. `List.jsp` is the JSP page that displays the types of products in the catalog, as shown in Figure 3.

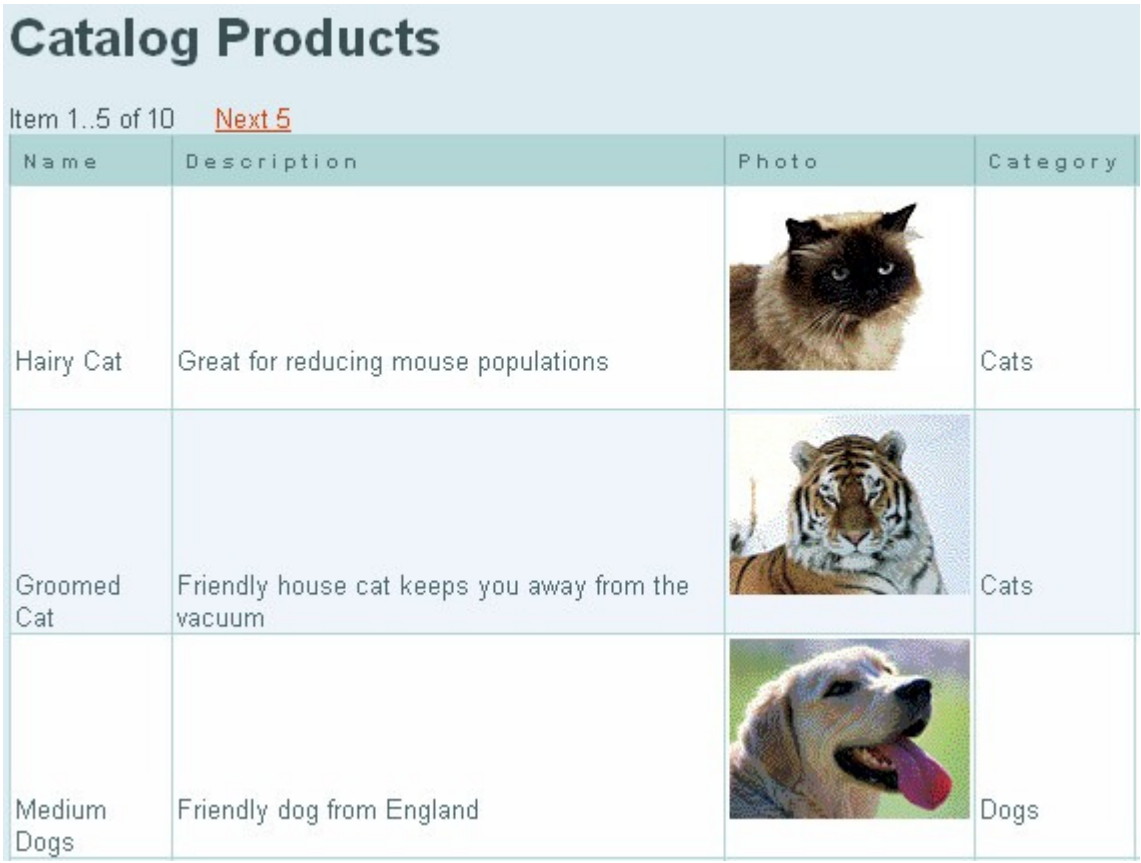

Figure 3. *Catalog Products Page*
Click the image to enlarge it and show an additional column.

Here is part of the code in `List.jsp`:

```
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<f:view>
  <html>
    ...
    <body>
      <h:messages errorStyle="color: red" infoStyle="color: green" layout="table"/>
      <h1>Catalog Products</h1>
      ...
          <h:dataTable value="#{product.productItems}" var="item1" border="0" cellpadc

            <h:column>
              <f:facet name="header">
                <h:outputText value="Name"/>
              </f:facet>
              <h:outputText value=" #{item1.name}"/>
            </h:column>
...
```

The tags prefixed by `f` or `h` are JavaServer Faces technology tags. Notice especially the `dataTable` tag. This tag is used to create a table and render it on the page. The tag is useful when you want to show a set of results in a table format. In an application that uses JavaServer Faces technology, the `UIData` component, the superclass of `dataTable`, supports binding to a collection of data objects. It does the work of iterating over each record in the data source -- in this case, the catalog. The

HTML renderer for `dataTable` displays the data as an HTML table.

The `value` attribute of the `dataTable` tag references the data to be included in the table. The `var` attribute specifies a name that is used by the components within the `dataTable` tag as an alias to the data referenced in the `value` attribute. Here, the `value` attribute points to a list of items in the `product` table in the catalog. The `var` attribute points to a single item in that list. As the `UIData` component iterates through the list, each reference to `item1` points to the current item in the list.

Notice too that the name, description, photo, and category of each item -- as well as show, edit, and destroy action links -- are displayed with JavaServer Faces technology `column` tags. The `column` tags represent columns of data in a `UIData` component. While the `UIData` component iterates over the rows of data, it processes the `UIColumn` component associated with each column tag for each row in the table. The `UIData` component iterates through the list of items, `product.productItems`, and displays the names, photos, and prices. Each time `UIData` iterates through the list of items, it renders one cell in each column.

The `dataTable` and `column` tags use `facet` tags to represent parts of the table that are not repeated or updated. These include headers, footers, and captions.

Building the Application

At this point, you might think that the CRUD application is fairly complicated and requires a lot of coding. As you build the application, you'll see that a lot of the coding is automatically generated by the NetBeans IDE.

A lot of the coding for the CRUD application is automatically generated by the NetBeans IDE.

Here's how to build the application.

1. If you haven't already done so, download and install NetBeans IDE 6.5, GlassFish v2UR2, and MySQL Community Server 5.1. You can download and install GlassFish v2UR2 with NetBeans IDE 6.5 as a single bundle.
2. Start the NetBeans IDE.
3. Ensure that GlassFish is registered in the NetBeans IDE, as follows:
   - Click the Services tab in the NetBeans IDE.
   - Expand the Servers node. You should see GlassFish v2 in the list of servers. If not, register GlassFish v2 as follows:
     - Right-click the Servers node and select Add Server. This opens an Add Server Instance wizard.
     - Select GlassFish v2 in the server list of the wizard and click the Next button.
     - Enter the location information for the server and click the Next button.
     - Enter the admin name and password and click the Finish button.
4. Start the MySQL database as follows:
   - Click the Services tab in the NetBeans IDE.
   - Expand the Databases node. You should see the MySQL server database in the list of databases, as shown in Figure 4.

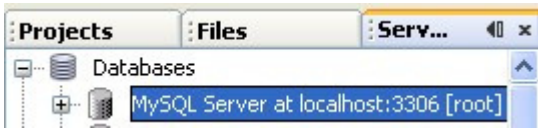

Figure 4. *MySQL Server Database in the Database List*

Note that the MySQL Connector/J JDBC Driver, which is necessary for communication between Java platforms and the MySQL database protocol, is included in NetBeans IDE 6.5.

The MySQL Connector/J JDBC Driver, which is necessary for communication between Java platforms and the MySQL database protocol, is included in the NetBeans IDE.

5. Set the properties of the MySQL server database as follows:
   - Right-click on the MySQL server database and select Properties. This opens the MySQL Server Properties dialog box, as shown in Figure 5.

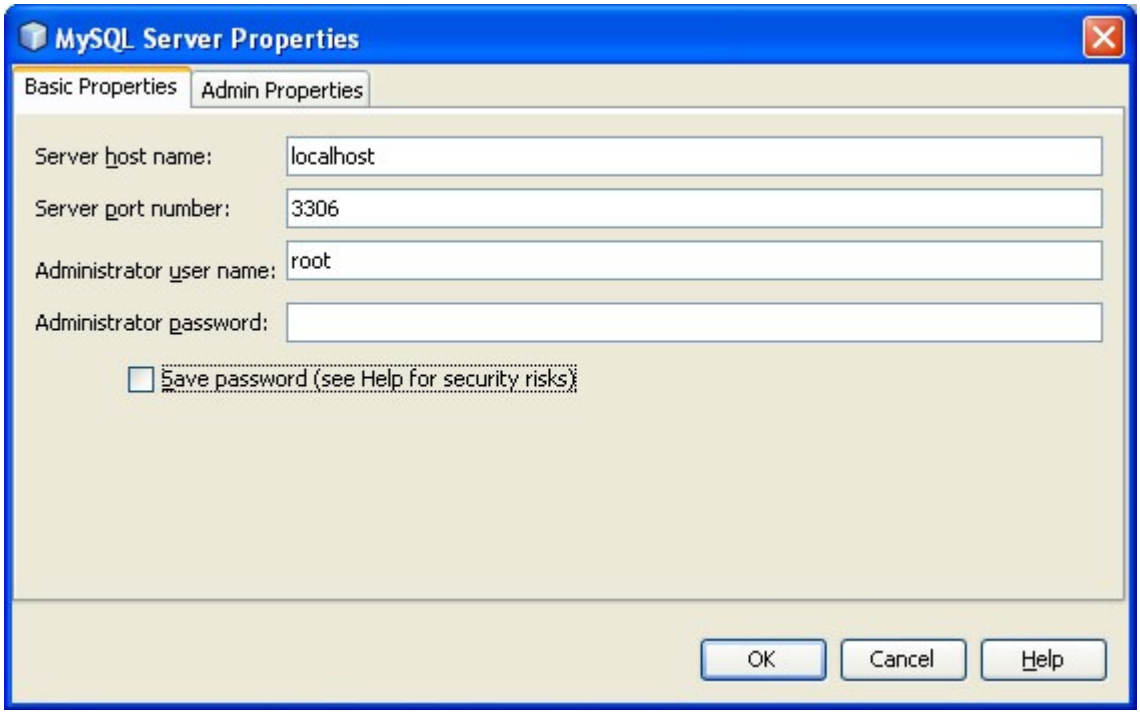

Figure 5. *MySQL Server Basic Properties*

- In the Basic Properties tab, enter the server host name and port number. The IDE specifies `localhost` as the default server host name and `3306` as the default server port number.
- Enter the administrator user name, if not displayed, and the administrator password, the default administrator password is blank.
- Click the Admin Properties tab.
- Enter an appropriate path in the Path/URL to admin tool field. You can find the path by browsing to the location of a MySQL Administration application such as the MySQL Admin Tool.
- Enter an appropriate path in the Path to start command. You can find the path by browsing to the location of the MySQL start command. To find the start command, look for `mysqld` in the `bin` folder of the MySQL installation directory.
- Enter an appropriate path in the Path to stop command field. You can find the path by browsing to the location of the MySQL stop command. This is usually the path to `mysqladmin` in the `bin` folder of the MySQL installation directory. If the command is `mysqladmin`, in the Arguments field, type `-u root stop` to grant root permissions for stopping the server. The Admin Properties tab should look similar to Figure 6.

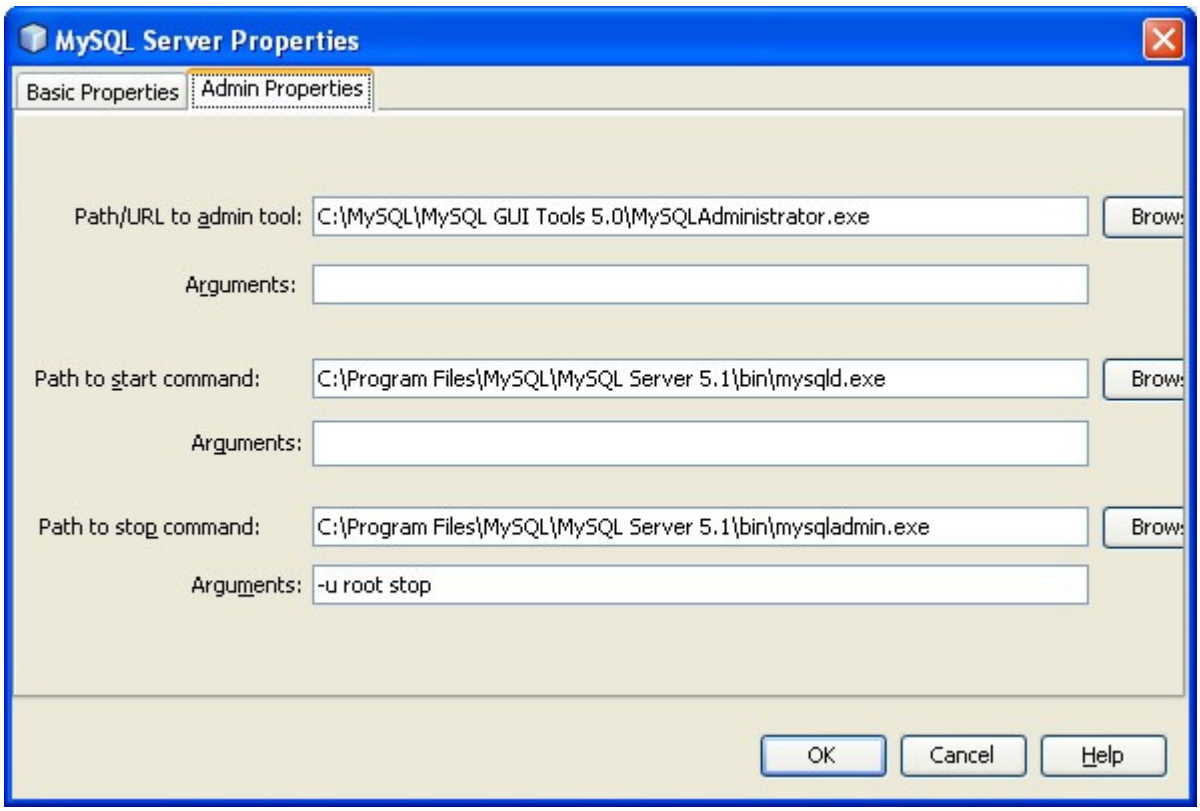

Figure 6. *MySQL Server Administration Properties*

- Click the OK button.

6. Right-click on the MySQL server database and select Start.
7. Create the `petcatalog` database as follows:
   - Right-click on the MySQL server database and select Create Database.
   - Enter the database name `petcatalog`. This will open a New Database Connection dialog box. Click OK to accept the displayed settings.
8. Create the tables in the MySQL pet-catalog database as follows:
   - Expand the Drivers node. You should see a driver for the `petcatalog` database in the list of drivers, as shown in Figure 7.

Figure 7. *Driver for the `petcatalog` Database*

   - Right-click the petcatalog driver and select Connect.
   - Right-click the petcatalog driver and select Execute Command. This will open an SQL command window.
   - Copy the contents of the `catalog.sql file` and paste the contents into the SQL editor, as shown in Figure 8.

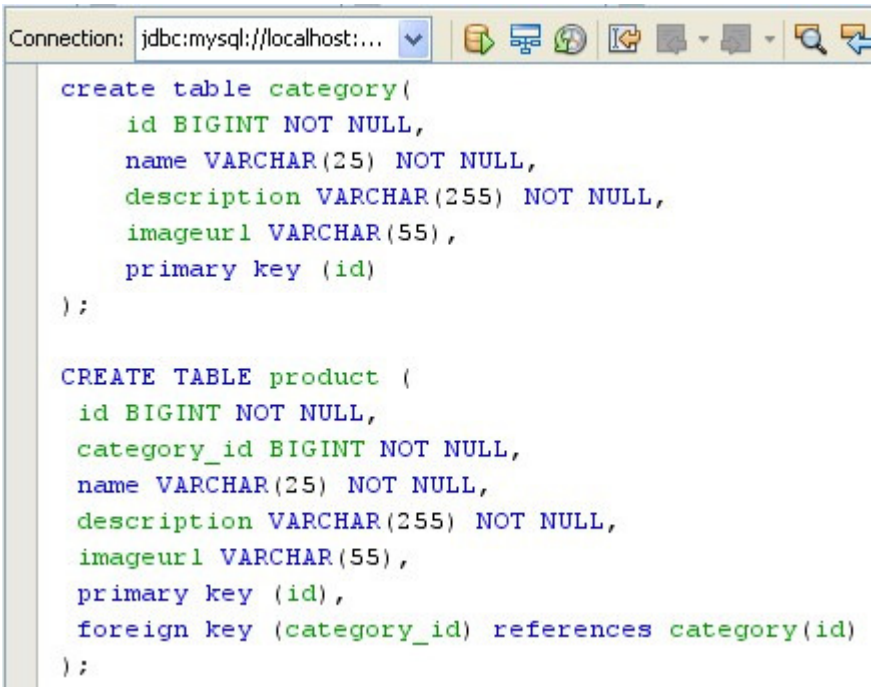

Figure 8. *Creating Tables in the Database*

   - Click the Run SQL icon  (Ctrl+Shift+E) in the SQL toolbar above the SQL editor.

     You can view the tables you just created. Right-click the Tables node below the `petcatalog` database in the Services window. You should see the database tables under the Tables node. You can expand a table node to see the table columns, indexes, and any foreign keys, as shown in Figure 9.

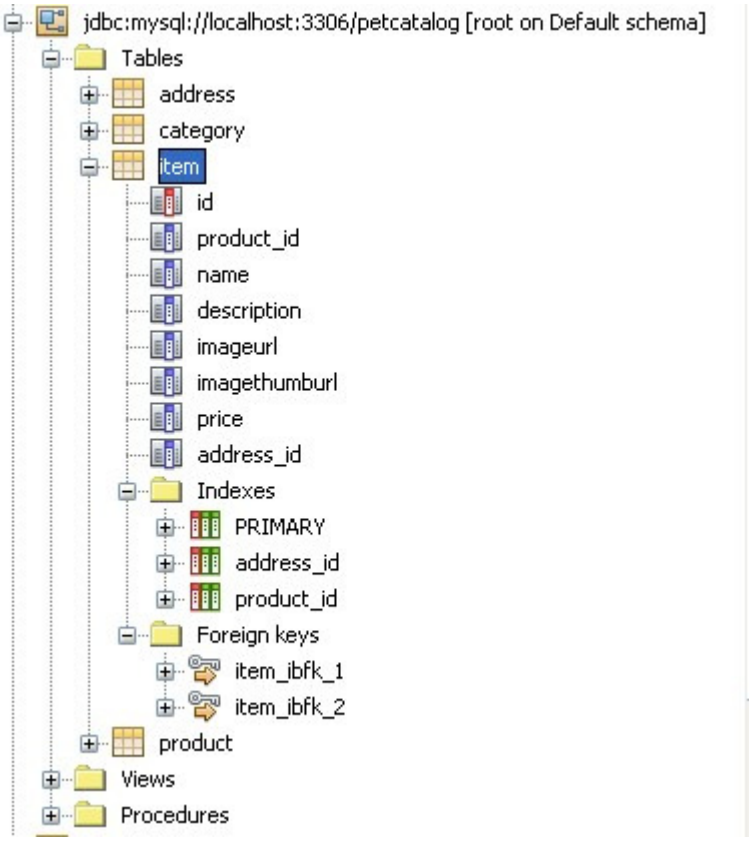

Figure 9. *An Expanded Table Node*

You can view the contents of a table or column by right-clicking the table or column and selecting View Data as shown in Figure 10.

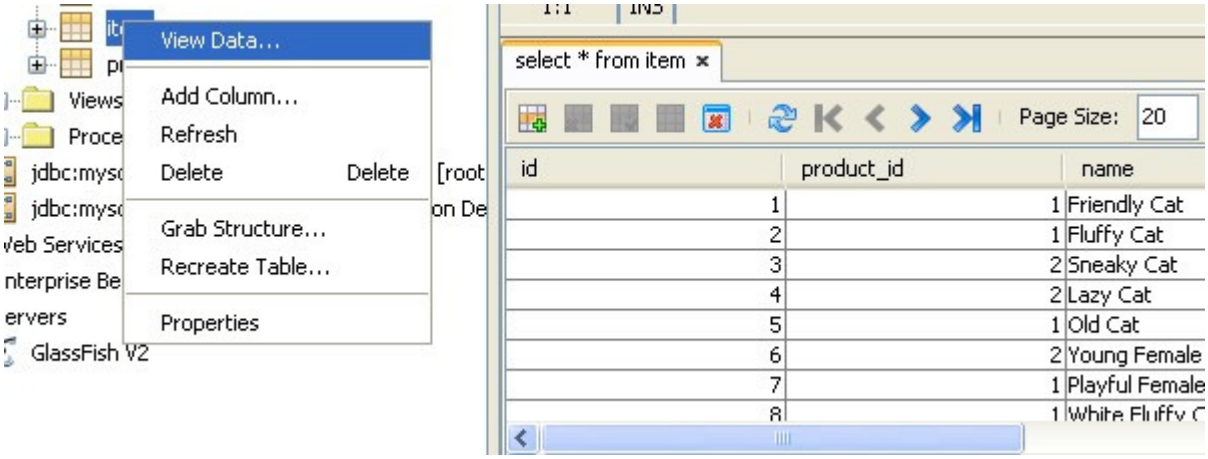

Figure 10. *Viewing the Contents of a Table*

9. Create the web application project as follows:
   - Select New Project (Ctrl-Shift-N) from the File menu in the NetBeans IDE. This opens the New Project window.
   - In the New Project Window, select Java Web from the Categories list and Web Application from the Projects list. Click the Next button. This opens the Name and Location page of the New Project window.
   - Enter `petcatalog` in the Project name field and a project location in the Project Location field. Check the Set as Main Project checkbox. Click the Next button. This opens the Server and Settings page of the New Project window.
   - Specify GlassFish as the server and select Java EE 5 as the Java EE version. Click the Next button. This opens the Frameworks page of the New Project wizard.
   - Check the JavaServer Faces checkbox in the Frameworks panel. Click the Finish button.

   In response, the IDE creates the basic structure for the web application project, including an `src` directory to hold the Java source files for the project, a `web` directory to hold the JSP and JavaServer Faces technology files for the project, and an `nbproject` directory to hold the metadata and build-related files for the project.

10. Add the pet images to the project. Copy the `images` folder from the `web` directory in the completed `petcatalog` application to the `web` directory in the web application that you are building. You should see images in the `web\images` folder such as the African spurred tortoise shown in Figure 11.

Figure 11. *African Spurred Tortoise*

11. Generate entity classes from the database. One of the powerful features of NetBeans IDE 6.5 is that it enables you to quickly create Java Persistence API entity classes from tables in a database. Here's how to generate entity classes from the tables that you created in the MySQL database:

One of the powerful features of the NetBeans IDE is that it enables you to quickly create Java Persistence API entity classes from tables in a database.

- In the Projects tab, right-click the `petcatalog` project node for the application and select New. Then select Entity Classes from Database. This opens the New Entity Classes from Database wizard.
- Select New Data Source from the Data Source drop-down list. This opens the Create Data Source dialog box.
- In the Create Data Source dialog, enter `jdbc/petcatalog` in the JNDI Name field and select `jdbc:mysql://localhost:3306/petcatalog` from the Database Connection drop-down list. Your entries should look like those in Figure 12.

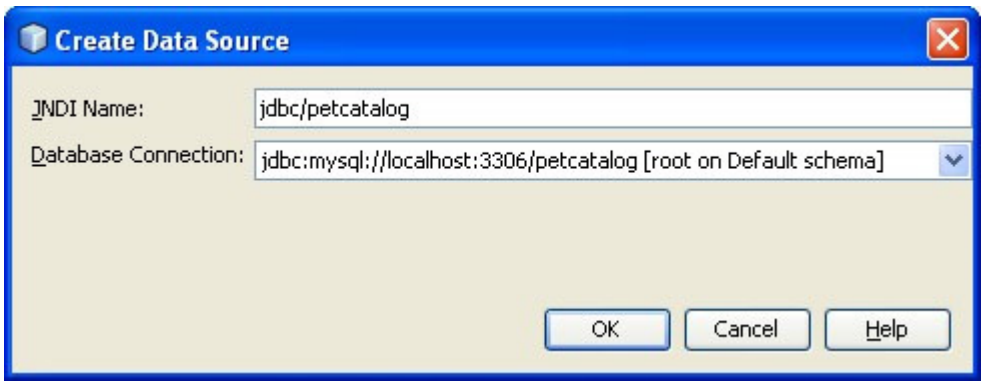

Figure 12. *Creating a Data Source*

- Click the OK button to close the dialog box and display the Database Tables page of the wizard. The tables in the `petcatalog` database appear in the Available Tables listbox, as shown in Figure 13.

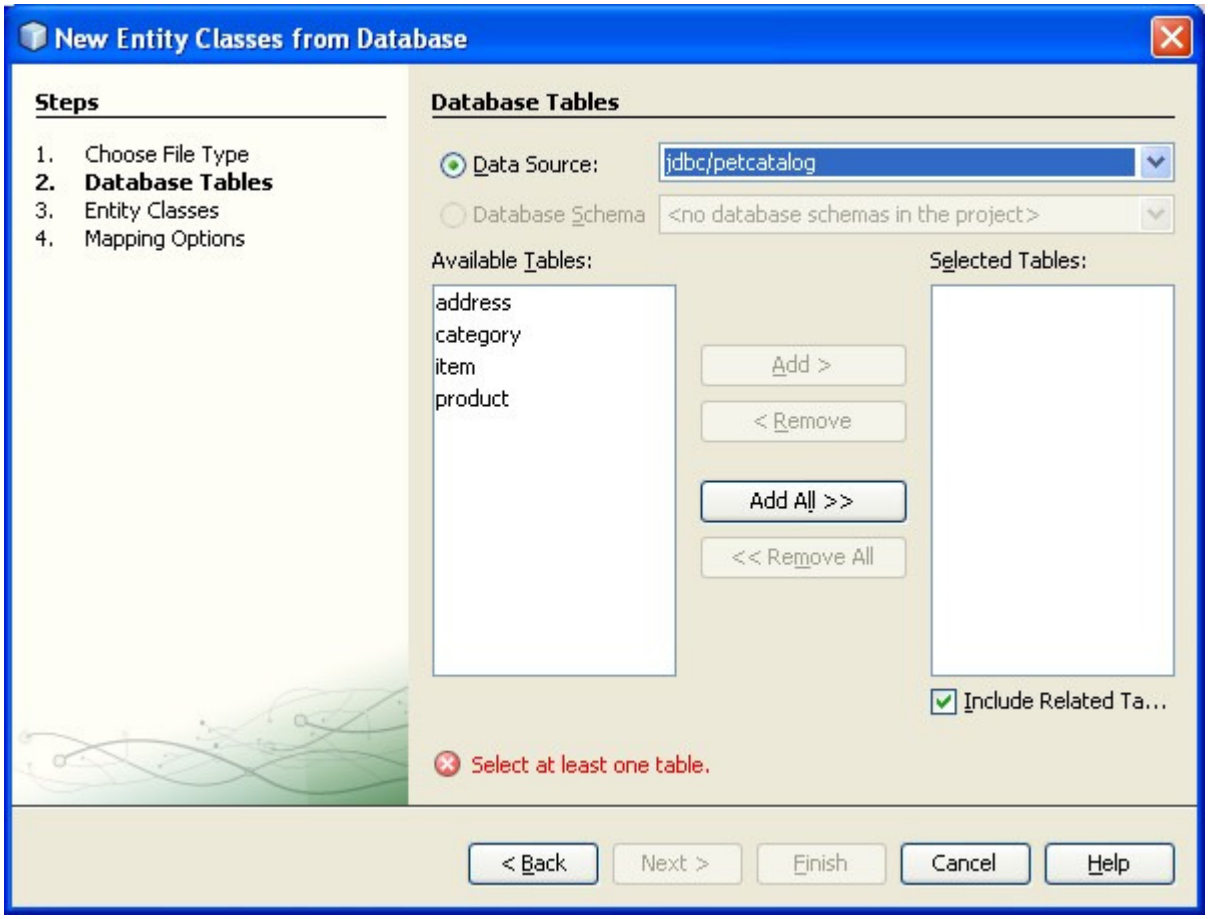

Figure 13. *Selecting Tables to Generate Entity Classes*

- Click the Add All button in the Database Tables page. This moves the tables in the Available Tables listbox to the Select Tables listbox. Click the Next button. This opens the Entity Classes page of the New Entity Classes from Database wizard.
- Enter `model` in the Package field. Make sure that the checkbox labeled Generate Named Query Annotations for Persistent Fields is selected. The Database Tables page should look as shown in Figure 14.

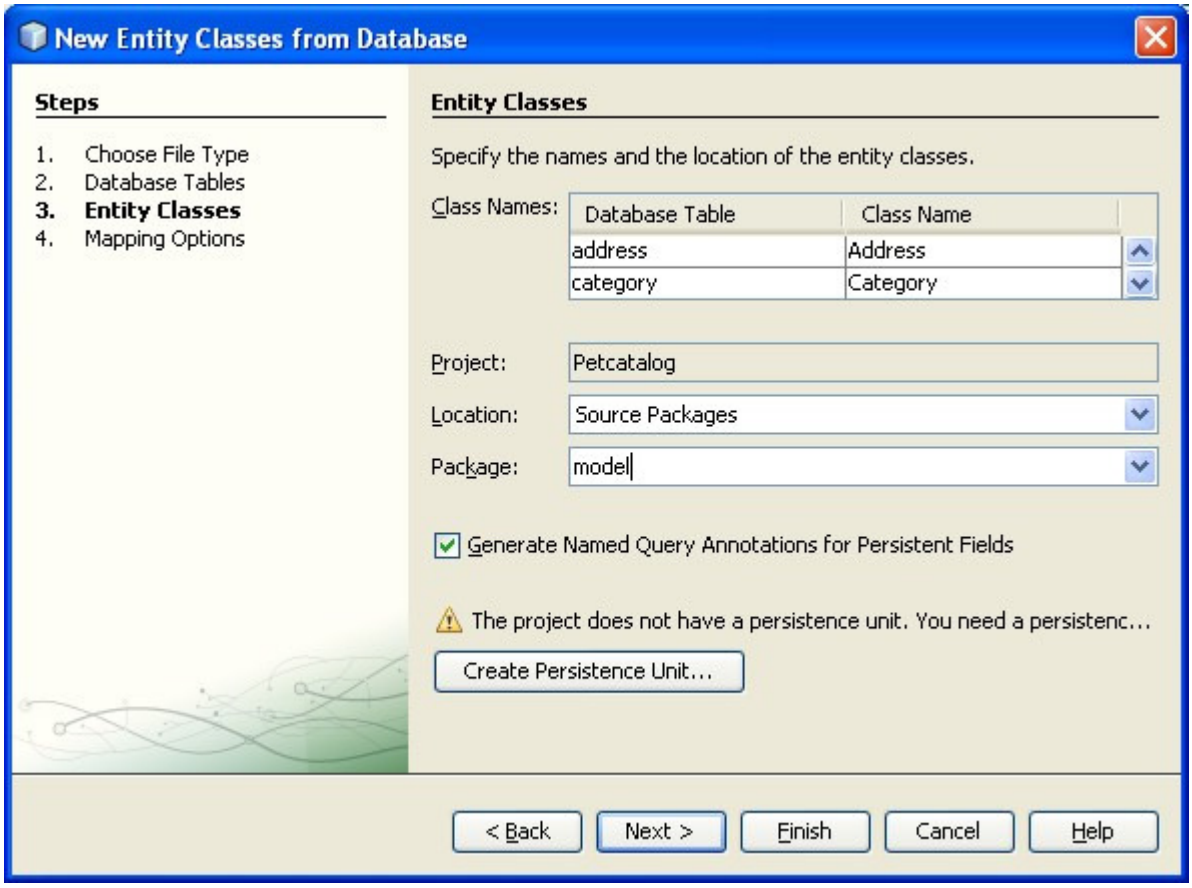

Figure 14. *Specifying Tables and Entity Classes*

- Click the Create Persistence Unit button. This opens the Create Persistence Unit dialog box. Leave the entries as is and click the Create button in the dialog box to create the persistence unit and return to the Database Tables page of the wizard.
- Click the Next button to open the Mapping Options page of the wizard.
- Leave the default values in the Mapping Options page as is and click the Finish button.

In response, the IDE generates an entity class for each table in the `petcatalog` database. You can see this in the Projects tab of the IDE by expanding the Source Packages node for the `petcatalog` project and then expanding the `model` node. This is shown in Figure 15.

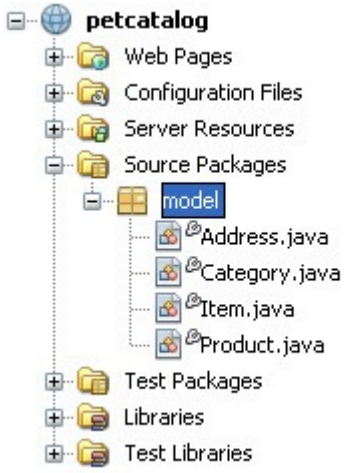

Figure 15. *New Entities Generated from Database Tables*

If you open each of the generated classes, you'll see that the IDE generated the Java code for the classes including all the needed annotations.

12. Edit the entity classes. Although the IDE generates the entity classes for the application, the content of each class needs to be slightly modified to fit the requirements of the application. For example, in `Address.java`, the entity class for the `address` table, the `@OneToMany` annotation specifies a `MappedTo` parameter value of `addressId`. In this application,

the value needs to be changed to `address`. In `Item.java`, the entity class for the `item` table, the `@ManyToOne` annotations specify the parameter `(optional = false)`. This specifies that a non-null relationship must always exist in the relationship. For this application, let's not make that restriction. You can either change the `@ManyToOne` annotations to specify a parameter of `(optional = true)` or not declare the `optional` parameter because a value of `true` is the default.

Change each of the entity classes as follows:

- `Address.java`:

  Change:

  ```
  @OneToMany(cascade = CascadeType.ALL, mappedBy = "addressId")
  ```

  To:

  ```
  @OneToMany(cascade = CascadeType.ALL, mappedBy = "address")
  ```

- `Category.java`:

  Change:

  ```
  @OneToMany(cascade = CascadeType.ALL, mappedBy = "categoryId")
  ```

  To:

  ```
  @OneToMany(cascade = CascadeType.ALL, mappedBy = "category")
  ```

- `Item.java`:

  Change:

  ```
  @JoinColumn(name = "address_id", referencedColumnName = "id")
  @ManyToOne(optional = false)
  private Address addressId;

  @JoinColumn(name = "product_id", referencedColumnName = "id")
  @ManyToOne(optional = false)
  private Product productId;
  ```

  To:

  ```
  @JoinColumn(name = "address_id", referencedColumnName = "id")
  @ManyToOne
  private Address address;

  @JoinColumn(name = "product_id", referencedColumnName = "id")
  @ManyToOne
  private Product product;
  ```

  Change:

```
public Address getAddressId() {
    return addressId;
}

public void setAddressId(Address addressId) {
    this.addressId = addressId;
}

public Product getProductId() {
    return productId;
}

public void setProductId(Product productId) {
    this.productId = productId;
}
```

To:

```
public Address getAddress() {
          return address;
}

public void setAddress(Address address) {
   this.address = address;
}

public Product getProduct() {
   return product;
}

public void setProduct(Product product) {
   this.product = product;
}
```

- Product.java:

Change:

```
@ManyToOne(optional = false)
    private Category categoryId;
@OneToMany(cascade = CascadeType.ALL, mappedBy = "productId"
```

To:

```
@ManyToOne
    private Category category;
@OneToMany(cascade = CascadeType.ALL, mappedBy = "product")
```

Change:

```
public Category getCategoryId() {
    return categoryId;
}

public void setCategoryId(Category categoryId) {
    this.categoryId = categoryId;
}
```

To:

```
public Category getCategory() {
    return category;
}

public void setCategory(Category category) {
this.category = category;
}
```

Make sure to save all your changes.

13. Generate JavaServer Faces technology pages from the entity classes. NetBeans IDE 6.5 not only enables you to quickly create Java Persistence API entity classes from tables in a database, it also enables you to generate JavaServer Faces technology pages, that is, JSP pages with JavaServer Faces technology components, from the entity classes. Here is how to generate JavaServer Faces technology pages from the entity classes:

The NetBeans IDE enables you to easily generate JavaServer Faces technology pages, that is, JSP pages with JavaServer Faces technology components, from entity classes.

- Right-click the `petcatalog` project node in the Projects window and select New. Then select JSF Pages from Entity Classes. This opens the New JSF Pages from Entity Classes wizard, as shown in Figure 16.

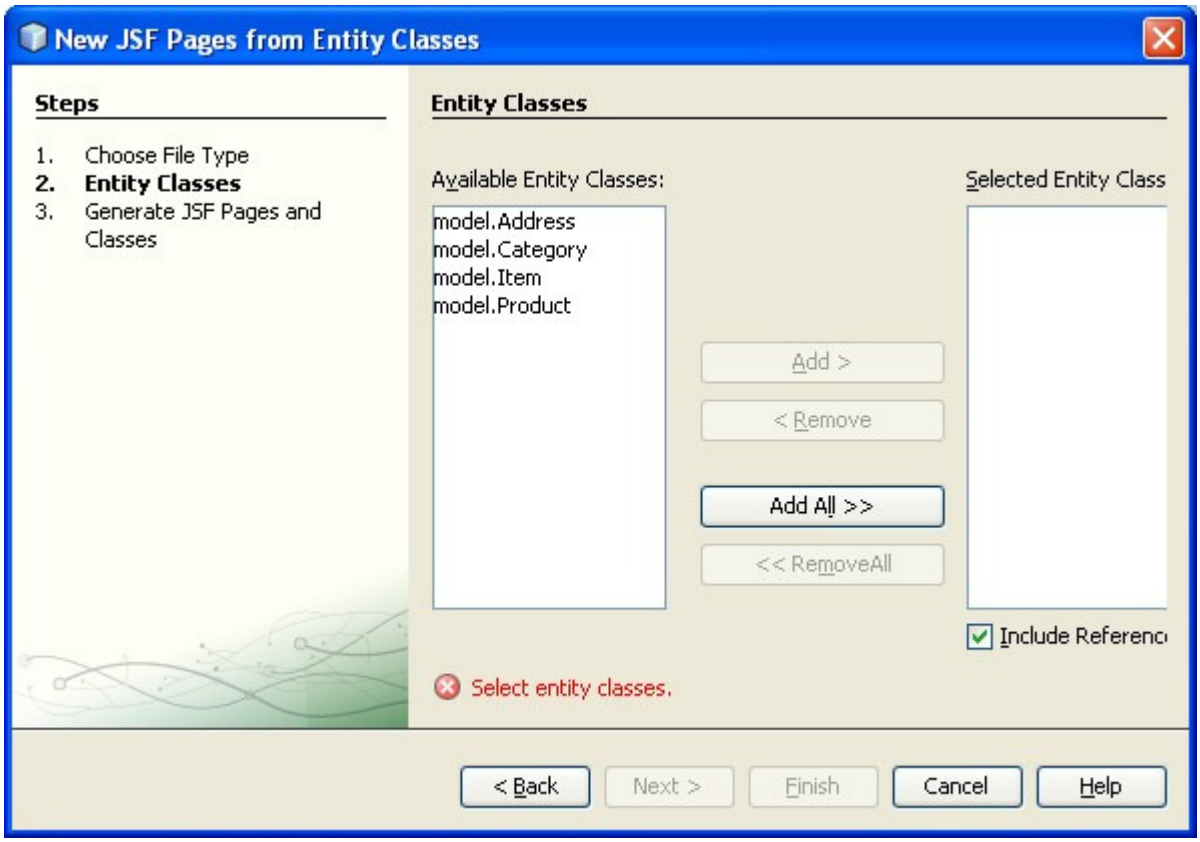

Figure 16. *New JSF Pages From Entity Classes*

- Click the Add All button to move all the entity classes to the Select Entity Classes pane of the wizard.
- Click the Next button to open the Generate JSF Pages and Classes page of the wizard.
- Specify `controller` in the JPA Controller Package field and ensure that `model` is specified in the JSF Classes Package field, as shown in Figure 17.

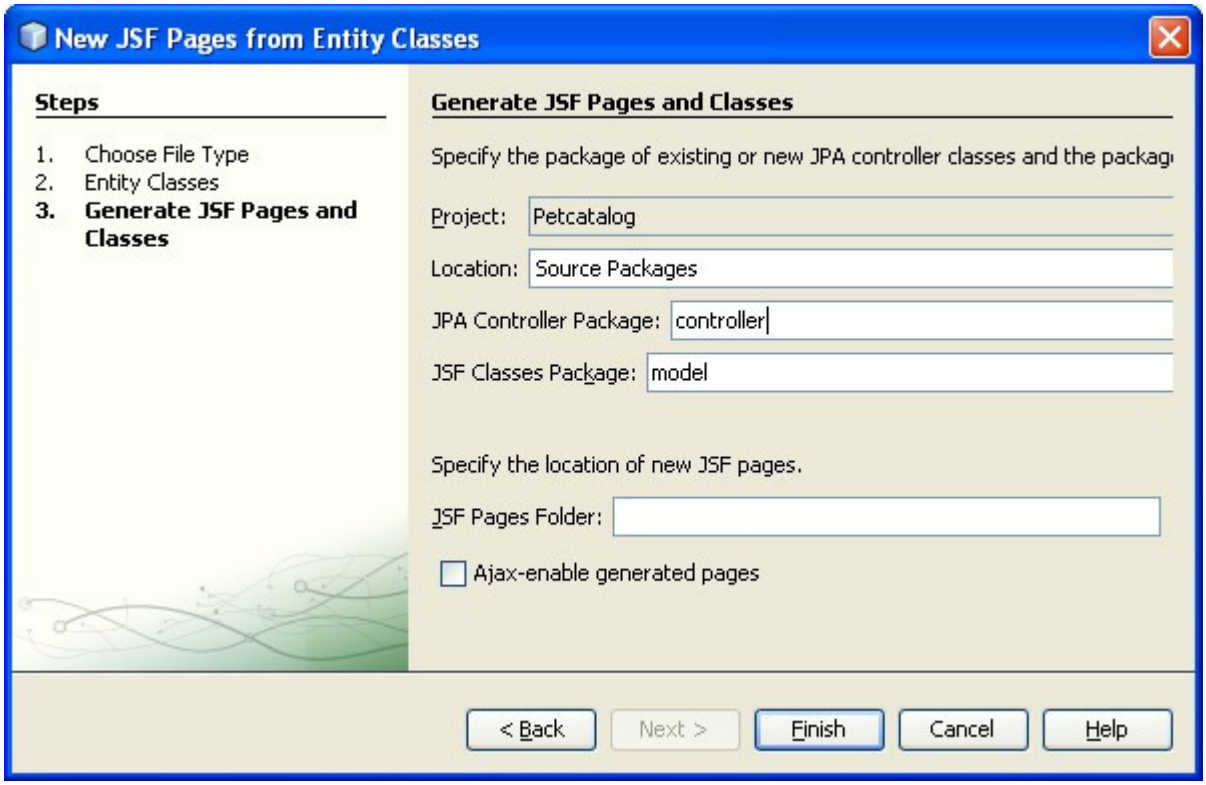

Figure 17. *Identifying the Controller and Model*

- You have the option to select the checkbox labeled Ajax-enable generated pages.
- Click the Finish button. In response, the IDE creates a `controller` package and generates Java Persistence API controller classes in that package for each entity class in the `petcatalog` project. The IDE also creates a `controller.exceptions` package and generates in that package exception classes for entity-related exceptions.

  In addition, the IDE generates JavaServer Faces technology converter and JavaServer Faces technology controller classes in the `model` package. It also creates a `model.util` package and generates a number of utility classes such as a JSF Expression Language resolver in that package .
  Expand the Source Package node in the `petcatalog` project to see the new classes. This is shown in Figure 18.

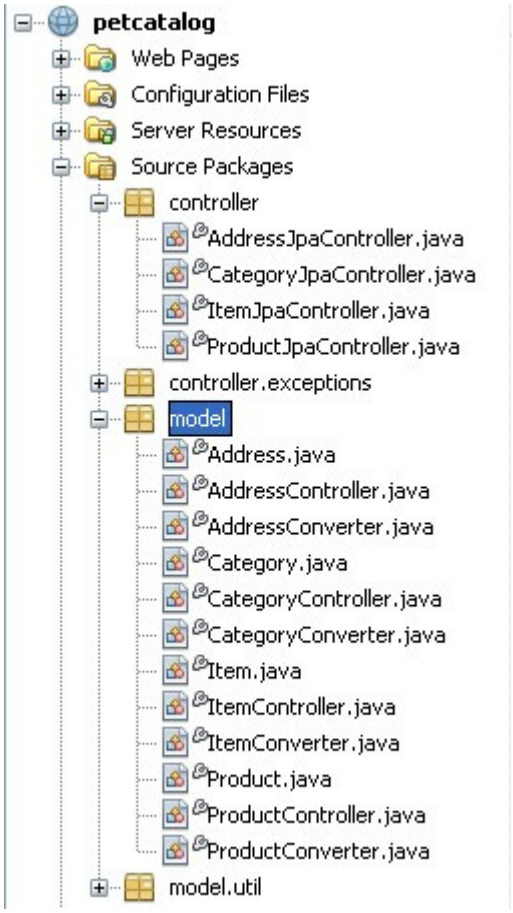

Figure 18. *Entity and Controller Classes*

Examine the contents of the Java Persistence API controller classes, JavaServer Faces technology converter classes, and JavaServer Faces technology controller classes. You'll see that the contents match those of the analogous classes in the completed `petcatalog` application.

If you expand the Configuration Files node and open the `faces-config.xml` file in the XML editor as shown in Figure 19, you'll see that the IDE inserted `<managed-bean>` and `<converter>` elements for each of the controller and converter classes. The IDE also inserted a `<navigation-rule>` element for each JSP page, indicating the outcome that causes the application to navigate to that JSP page.

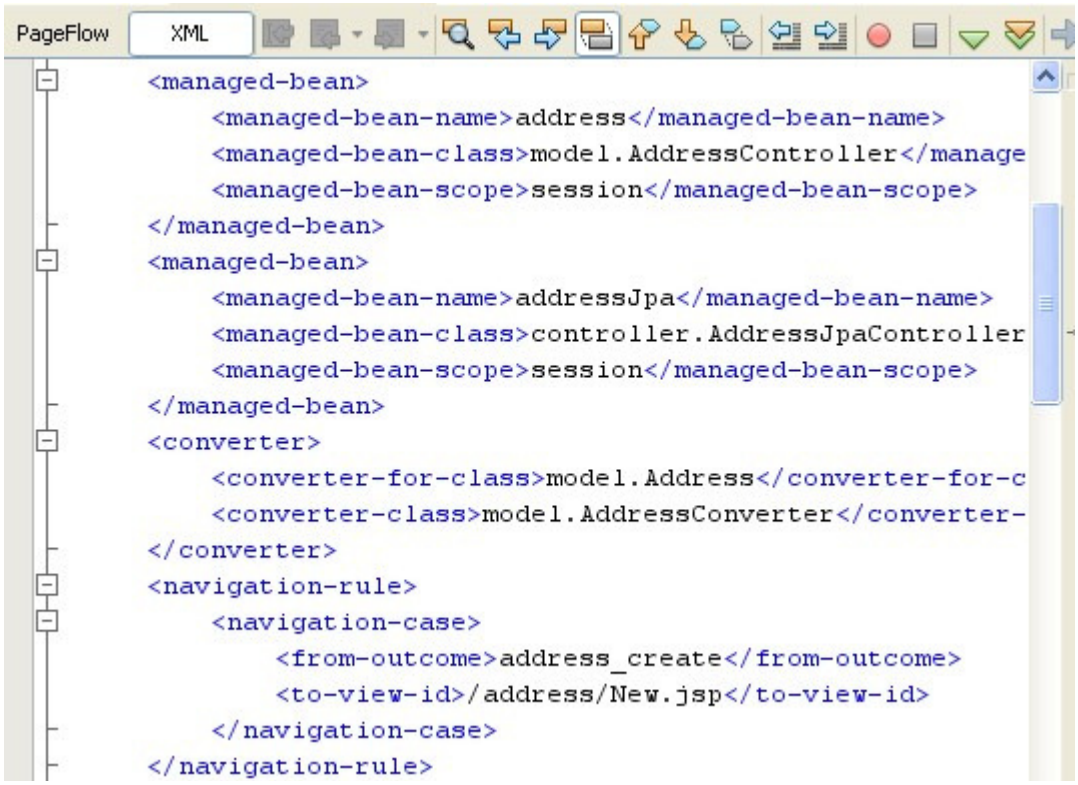

Figure 19. *The `faces-config.xml` File*

14. Edit the JSP pages. If you expand the Web Pages node, you'll see that the IDE also generated a folder for each of the entity classes. Each folder contains the files `Detail.jsp`, `Edit.jsp`, `List.jsp`, and `New.jsp`. The IDE also modified the `welcomeJSF.jsp` file by inserting links to each of the `List.jsp` pages. The IDE also created a Script file, `jsfcrud.js`, and a CSS file, `jsfcrud.css`. The Script file is used primarily to handle Ajax-related actions if you selected the Ajax-enable generated pages checkbox in the New JSF Pages from Entity Classes wizard. The CSS file controls the style attributes, such as color and font, used in the web pages.

    Although the contents of the JSP files are close to what the application requires, they still require modification. For example, the `welcomeJSF.jsp` page will display the heading "JavaServer Faces." You want it to display "Pet Catalog." In addition, the `Detail.jsp` and `List.jsp` pages in the entity classes folders do not currently display the exact set of columns to be displayed.

    Here's what to change in the `welcomeJSF.jsp` page:

    Change:

    ```
    <title>JSP Page</title>

    <h1><h:outputText value="JavaServer Faces" /></h1>
    ```

    To:

    ```
    <title>Pet Catalog</title>

    <h1><h:outputText value="Pet Catalog" /></h1>
    ```

    You must compare the `Detail.jsp` and `List.jsp` pages in each of the entity class folders with their equivalents in the finished `petcatalog` application to identify the required changes. For example, here are the changes to make to the `List.jsp` file in the `Item` folder:

    Change:

```
<title>Listing Item Items</title>

<h1>Listing Item Items</h1>

<h:column>
       <f:facet name="header">
              <h:outputText value="Id"/>
       </f:facet>
              <h:outputText value=" #{item1.id}"/>
</h:column>
<h:column>
       <f:facet name="header">
              <h:outputText value="Name"/>
       </f:facet>
              <h:outputText value=" #{item1.name}"/>
</h:column>
<h:column>
       <f:facet name="header">
              <h:outputText value="Description"/>
       </f:facet>
              <h:outputText value=" #{item1.description}"/>
</h:column>
<h:column>
       <f:facet name="header">
             <h:outputText value="Imageurl"/>
       </f:facet>
            <h:outputText value=" #{item1.imageurl}"/>
</h:column>
<h:column>
       <f:facet name="header">
             <h:outputText value="Imagethumburl"/>
       </f:facet>
             <h:outputText value=" #{item1.imagethumburl}"/>
       </h:column>
<h:column>
       <f:facet name="header">
           <h:outputText value="Price"/>
       </f:facet>
           <h:outputText value=" #{item1.price}"/>
</h:column>
<h:column>
       <f:facet name="header">
           <h:outputText value="Address"/>
       </f:facet>
           <h:outputText value=" #{item1.address}"/>
</h:column>
<h:column>
       <f:facet name="header">
           <h:outputText value="Product"/>
       </f:facet>
           <h:outputText value=" #{item1.product}"/>
</h:column>
```

To:

```
<title>Pet Catalog Items</title>

<h1>Pet Catalog Items</h1>

<h:column>
        <f:facet name="header">
                <h:outputText value="Name"/>
        </f:facet>
                <h:outputText value=" #{item1.name}"/>
</h:column>
<h:column>
        <f:facet name="header">
                <h:outputText value="Photo"/>
        </f:facet>
                <h:outputText value=" #{item1.imagethumburl}"/>
</h:column>
<h:column>
        <f:facet name="header">
                <h:outputText value="Price"/>
        </f:facet>
                <h:outputText value=" #{item1.price}"/>
</h:column>
```

15. Deploy and run the project by right-clicking the `petcatalog` project and then selecting Run. You can also click the Run Main Project (F6) icon  in the main toolbar or click the Run menu item.

    In response, the IDE saves all the changed files, rebuilds the application, starts the GlassFish v2UR2 application server, and deploys the application to the application server. Your default web browser opens to the URL `http://localhost:8080/petcatalog/` and displays the Pet Catalog welcome page as shown in Figure 20.

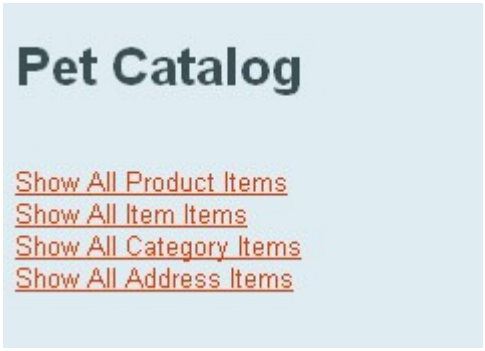

Figure 20. *Pet Catalog Welcome Page*

Try out some of the links. For example, if you click the "Show All Product Items" link, you should see the Catalog Products page shown in Figure 3.

Summary

GlassFish fully supports Java EE 5 web technologies, including dynamic web technologies such as JSP and JavaServer Faces technology. In addition, GlassFish has a built-in persistence manager, the Java Persistence API, and supports ease-of-use features such as annotations. These capabilities, combined with the simplicity of querying and manipulating MySQL databases, make it easy to build a web application with data persistence that accesses a MySQL database and deploy it on a GlassFish application server. The NetBeans IDE, which integrates MySQL and GlassFish, makes building these applications even easier.

In the next article in this series, you'll see how easy it is to create a web application that accesses a web service through GlassFish's support for the Metro web services stack and updates a MySQL database.

For More Information

- GlassFish and MySQL, Part 1: A Perfect Combination For Web Applications
- GlassFish and MySQL, Part 3: Creating and Using a Web Service
- GlassFish and MySQL, Part 4: Creating a RESTful Web Service and JavaFX Client
- MySQL
- GlassFish
- Java EE Technologies
- The Aquarium

Rate and Review

Tell us what you think of the content of this page.

Excellent **Good** **Fair** **Poor**

Comments:

Your email address (no reply is possible without an address):
Sun Privacy Policy

Note: We are not able to respond to all submitted comments.

Submit »